Graphík Programming Language

Complete documentation and language specification

1. Introduction to the language:

The Graphík programming language is an imperative, general-purpose(although this is a bit questionable), programming language. Its main(and only, as of now) implementation is the Graphík Lightweight Compiler, also known as GLC for short. Graphík is a compiled language(obvious, by the fact that its main implementation is a compiler). Its main purpose is to be used as an additional utility for the Graphík Graphical Calculator, as its only output capability is to print two integers on the command line which will basically serve as coordinates to be `plotted` on the calculator's Cartesian plain. The calculator and the compiler are two different applications though, and the compilation process of a Graphík program will result in an independent application, utilizing Graphík's main purpose is not exactly `crystal clear`. We will be talking about interaction between the calculator and a Graphík application in the following paragraphs.

2.Language specification:

-Every expression is evaluated as a 64 bit signed integer.

-The entry point of a standard-compliant Graphík program bears the identifier `main`, which takes no arguments.

-Function arguments are pushed on the stack.

-All functions must have a return value(otherwise, assembly instructions for returning from a function are not generated, causing a segmentation fault).

3.Grammar overview:

Graphík's grammar is quite similar to the one of the Flex++ Scripting Language. However, the two languages do not have much in common, despite common keywords and grammatical rules. Flex++ is strictly an interpreted language(although a compiler can also be built for it), while Graphík is a compiled language with a much more complex and extended grammar.

The Graphík programming language, as of now, includes 10 keywords:

---

begin, end, declare, none, while, if, ret, break, plot, invoke

---

It supports the following arithmetic and logical operations:

+ -

* /

!= ==

&& ||

> <

>= <=

A) Order of operations:

Arithmetic operations are performed in the same order as in standard mathematics. All of the logical operators have the same precedence, therefore an expression, similar to:

<expr> >= <expr> || <expr> != <expr>

should be written with parentheses:

(<expr> >= <expr>) || (<expr> != <expr>)

B) Function declaration:

A function declaration and a function definition are one and the same thing in Graphík. Possibly, there will be a workaround this particular rule, since it basically renders including external libraries in a Graphík program impossible, as the compiler will refuse to compile a program which includes a call to a function whose existence is not known to the compiler, and declaring the function in the Graphík source code will also make a local definition of that function. That will cause the compiler to call this function instead of the one, defined in the external library. Nevertheless, a function that takes no arguments looks like this:

declare func(none) begin

...

end

A function that takes arguments looks like this:

declare func(arg1 arg2 arg3) begin

...

end

Note that there are no commas inbetween the different arguments, only spaces are required. Using commas will result in a compile-time error.

The keywords `begin` and `end` are used to open and close blocks respectively. In this case, they are used for the function's scope. The `declare` keyword is used to declare a function with the identifier `func` which can either take no arguments(marked by the `none` keyword) or take a number of arguments, separated by a single space.

C) Variable definition and assignment:

If a variable is not defined, definition and assignment are equivalent. They obey the following pattern:

<identifier> "=" <expr>

Now the variable is defined and assigned to a certain expression. Following its definition, it now has an entry in the current program's symbol table. Further references to that variable in a similar manner will only be taken for assignment. Therefore, no new entries will be made for that particular variable in the symbol table.

Example:

```
declare main(none) begin
var = 3 + 4 * 2        <--- Definition
var2  = 34

var = var2             <--- Assignment

      ret 0
end
```

D) Conditional statements:

There are exactly two conditional statements in Graphík:

A loop and an if-statement.

```
while <expr> <block>
if  <expr> <block>
```

The condition of a conditional statement is an expression and such a statement should also possess a block to execute in case its condition evaluates to `true`.

Example:

while var2 + 15 < 66 / 3

begin

....

end

The if-statement looks identical to the `while` loop. A `break` statement may be used within a `while` loop. Usage of `break` outside of a loop won't cause a compile-time error, but it will result in undefined behaviour(or atleast, something the user does not expect in the least)

E) Expressions:

Every numeric value, variable, function call and operation is an expression in terms of Graphík's grammar. Function calls are done by using the `invoke` keyword, followed by the identifier of the function and the arguments, to be passed to the function.

invoke <ident>(<expr>{ <expr>})

As this is an expression, the following statement is true:

callres = 3 + 4 * invoke func(10 20 30)

IMPORTANT: *Function calls are expressions, but an expression cannot exist without being a part of a statement, in terms of Graphík's grammar, hence only invoking a function would not work. Because of that, we must always store the result of a function call in a variable, since assignment or variable definition is a statement, therefore we can have an expression within it. Calling a single function will look like this:*

retval = invoke func(a1 a2 a3)

Or: ***retval = invoke func()***

if the function does not take any arguments.

F) Plot statement:

The `plot` statement is a pretty simple construct. In its escence, it's merely a call to the C library function `printf`, given two positional arguments(not counting the format) - the X coordinate and the Y coordinate(from the calculator's perspective). In the future, Graphík will have its own standard library, hence deprecating the need of the C standard library. In conclusion, the `plot` statement takes two expressions and prints them in the standard output of the program, separated by a single space:

plot var + 3 var2 * 3

(for convenience) ***plot (var + 3) (var2 * 3)***

Output:

37 102

*(considering that this `plot` statement takes place after the assignment -> **var = var2**, that we showed earlier)*

G) Return statement:

The return statement is an obligatory part of every function in Graphík. As mentioned previously, not using it will make the compiler not generate `ret` instructions in the generated assembly, which will ultimately cause a segmentation fault in the least. It is followed by a single expression, which will be returned to the caller as the return value of the callee:

ret <expr>

Examples:

ret 3

ret var

ret var + 3

ret invoke func({args})

4. Interaction with GGC(Graphík Graphical Calculator):

The documentation is nearing its conclusion. In that case, it is finally time to speak about the Graphík programming language's main purpose – interacting with the graphical calculator. The easiest way to do this is to basically feed the output of a Graphík program into a text file and load it into the calculator. All of the points, defined by the coordinates in the text file will be plotted on the calculator's Cartesian plain, resulting the visualization you probably intended to achieve. This can simply be done from the command line:

```
glc –o a.exe src.gk

a.exe > out.txt
```

Now you simply load `out.txt` into the calculator and you get a graphical representation of your program's output.

5. This is it!

You are now ready to use and utilize the Graphík programming language for your own needs. Even though it's quite a small language, it can still be useful to you in case you want to make a quick graphical representation of a certain mathematical sequence or simply some kind of a geometric shape. The language can be used beyond that though, for all kinds of applications where a certain calculation should be made.